Eclipse Litepaper

Motivation

Eclipse offers customizable, modular rollups. As a rollup, Eclipse does not have its own Byzantine fault-tolerant consensus. Eclipse's modular design allows us to benefit from the high throughput of a fast execution layer while borrowing the security of a highly decentralized data availability layer.

We are initially deploying to Cosmos as a sovereign rollup (described under "High-Level Architecture"). Eclipse enables a variety of dApps on Cosmos that are not possible right now:

- Decentralized central limit order books (CLOBs)
- Streaming payments such as Zebec
- Games that utilize high TPS
- dApps that rely on CLOB infrastructure
- Trading real-world assets (RWAs) with sizeable oracle data
- Notifications and messaging
- Blockchain for Nasdaq, now on Cosmos

The Eclipse execution layer is forked from Solana's execution layer, which can support up to 700,000 TPS*. This includes features such as Solana's virtual machine (extended Berkeley Packet Filter or eBPF), parallel processing of transactions (Sealevel), and pipelining (transaction processing unit). For the Solana community, we bring the following benefits:

- Enable Solana dApps to go multichain and access inter-blockchain communication (IBC)
- App chains and app zones for Solana dApps to gain sovereignty and avoid congestion
- Custom sequencer logic such as block times, returning MEV to dApps, subsidizing fees
- Be a canary network (like Kusama on Polkadot) for the latest execution layer upgrades
- Easily spin up private chains that use the Solana execution layer
- Develop new execution layer upgrades that can be merged back into the Solana L1
- 1-to-1 cross chain invocation semantics between Solana and Cosmos

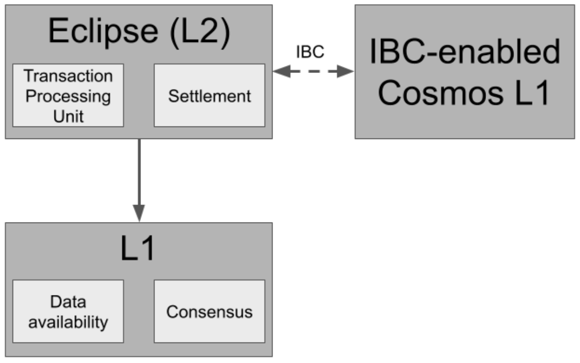
This document goes into implementation details and the long-term plan for developing Eclipse. See the announcement post for further motivation and the short-term plan.

High-Level Architecture

A typical, monolithic L1 involves (i) executing transactions, (ii) settling disputes between validators (if applicable), (iii) ordering transactions, and (iv) publishing finalized block data to the network.

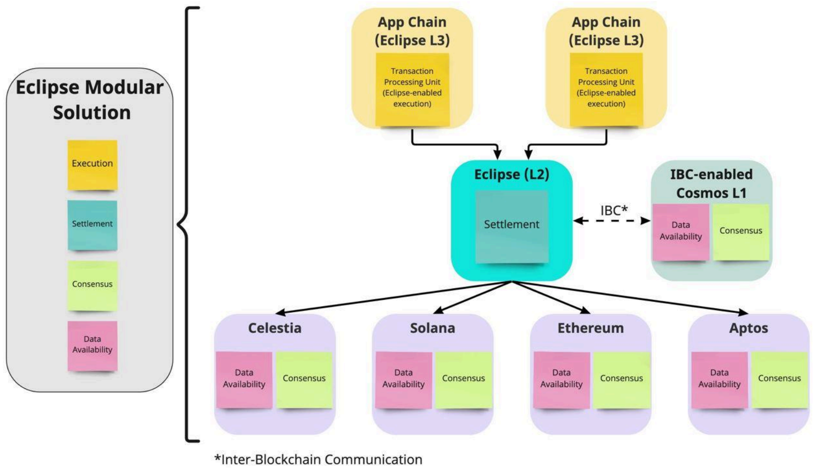
* The Solana L1 TPS is impacted by consensus messages, resulting in closer to 5,000 TPS.

Motivated by the Celestia project, the insight behind Eclipse's design is that these are four separate responsibilities. Eclipse will be responsible for (i) execution and (ii) settlement, while leaving (iii) consensus and (iv) data availability to the L1. By definition, this makes Eclipse a "sovereign rollup." This design allows us to leverage Solana's highly-optimized execution layer, while decoupling ourselves from the Solana consensus where stability issues are being resolved.



The choice of L1 here is somewhat flexible, but the L1 must be capable of handling the volume of data availability that we will be posting, which we estimate is 10 gigabytes per day at Solana's current transaction volume (July 2022). Since our execution layer is forked from Solana, we know that the Solana L1 can handle this volume of data availability. At launch, Celestia will support 17 gigabytes per day, which will increase over time.

In the future, we will allow users to choose where they post data availability, removing our dependency on any specific ecosystem. We will enable posting data availability to Ethereum once it is feasible to do so: post-Danksharding, Layr Labs, and other optimizations. We will further decouple the design by only handling settlement on the Eclipse L2, leaving execution to Eclipse app chains or app zones.



In this case, Eclipse becomes a "settlement rollup," enabling us to horizontally scale execution. We might continue to allow general purpose, permissionless execution on the Eclipse L2 to run small dApps until they require their own Eclipse app chain.

Execution Layer: Sealevel Virtual Machine (SVM)

The Solana GitHub repo does not separate the execution from consensus. Optimizations that we want to keep from the execution layer include Sealevel (parallel processing for transactions) and pipelining. Parts that we are interested in removing include Solana's Turbine block propagation and Gulf Stream transaction forwarding. Sharing the runtime enables Solana developers to use their existing suite of developer tools on Eclipse, such as deploying with the Solana CLI using `solana config set --url [eclipse_url]` then `solana program deploy [program_path]`.

We post blocks of data to Celestia via a Geyser plugin in our fork of the Solana repository. This design is compatible with posting data availability to other L1s in the future.

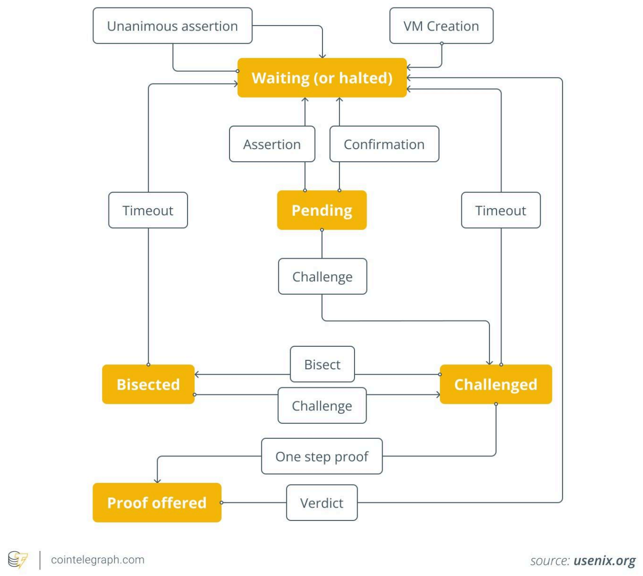
Settlement Layer

Whether this is an optimistic or zero-knowledge rollup, the engineering lift is high. As an optimistic rollup, we require evaluating BPF bytecode on-chain or implementing a bisection game (described below). As a zero-knowledge rollup, we must implement sophisticated zero-knowledge circuitry for the eBPF virtual machine, which has not been done before, in addition to handling some unsolved research questions. We discuss these two possible approaches below. The plan in the long-term is to provide a non-enshrined settlement layer for Eclipse execution chains. Eclipse starts as an optimistic rollup and will become a zero-knowledge rollup in the long run.

Optimistic Rollup

An optimistic rollup relies on an honest minority security model: at least one node for the L2 must be honest (acting rationally). The L2 executes transactions, and periodically a commitment to the current state of the rollup is posted on-chain to the L1. If this state is inaccurate, any actor can call for a "fraud proof," which comes in two flavors [citation]: non-interactive and interactive. There is a fixed challenge period during which anyone can call for a fraud proof.

A non-interactive fraud proof requires replaying a full transaction when a fraud proof is invoked. This is an expensive operation which led to various challenges for Optimism, motivating the design of the Optimism Virtual Machine (OVM), imposing limitations on contract size and bounding gas consumption. For V2, Optimism is moving closer to interactive fraud proofs [citation]. For these reasons, our optimistic rollup will use interactive fraud proofs.



The interactive fraud proof requires both participants to be online. Through a so-called "bisection game," the participants determine which instruction they disagree on in a logarithmic number of steps. The bisection game requires a third-party to verify the structure of each bisection. Once the instruction is determined, it is re-executed on-chain via a "virtual machine interpreter." For the sake of brevity, we leave details of the interactive fraud proof implementation to the Arbitrum whitepaper, since it is well-known.

The eBPF instruction set (and its Solana implementation) is more restrictive than the EVM instruction set. Therefore, we expect implementing the virtual machine interpreter for the eBPF should take less development time than for the EVM. If Rust has a target for the L1's virtual machine, we can leverage the rbpf or solana_rbpf library, although this might require the use of unsafe Rust. Alternatively, we can commit a zero-knowledge proof of this single instruction execution.

Pros of the optimistic rollup: The optimistic rollup has less technical uncertainty than the zero-knowledge rollup. There are no unsolved technical challenges. While the average user will have a 1-2 week time-to-finality due to the challenge period, any user can achieve fast finality by running their own node and verifying the validity of the block.

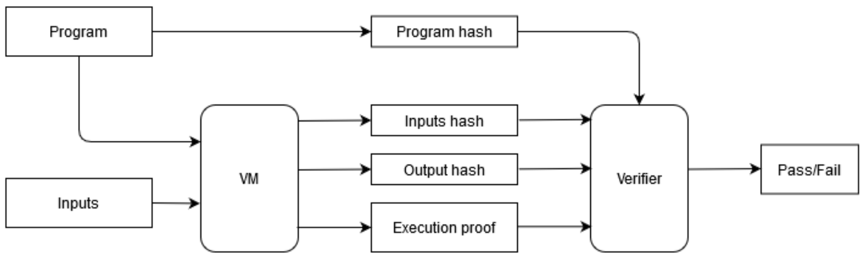
Cons of the optimistic rollup: The tradeoff is lesser portability and composability with other chains. Moving traditional optimistic settlement to another L1 would require re-implementing the virtual machine interpreter, while zero-knowledge settlement only requires implementing the verifier protocol (below). An alternative is a dedicated settlement layer. For both optimistic and zero-knowledge rollups, the sequencer can theoretically censor transactions from being included. For optimistic rollups, an economically irrational actor can temporarily stall the rollup through mistakenly calling for fraud proofs.

Zero-Knowledge Rollup

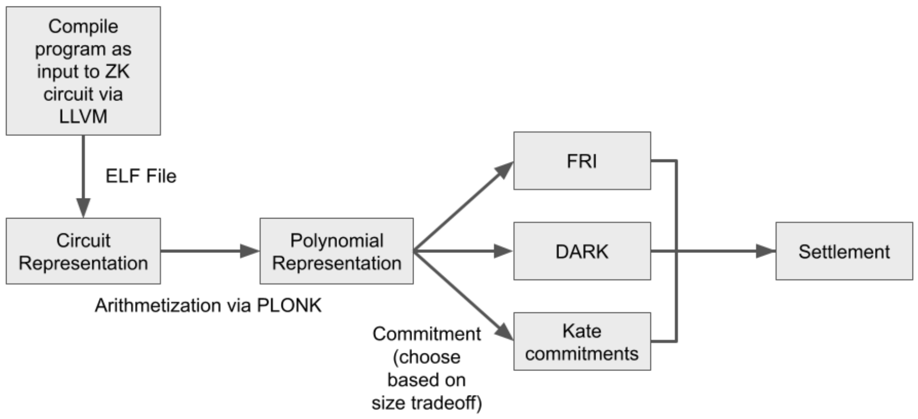
A zero-knowledge rollup does not rely on any node in the L2 to be honest nor rational. Instead, only those transactions that were provably executed correctly are accepted by the settlement layer via a zero-knowledge proof.

In general, to execute a given program, the program is formulated as a circuit or machine. Given an input, a prover produces a probabilistically checkable proof (PCP) that they have a solution to this circuit or machine and makes a vector or polynomial commitment. Finally, a verifier protocol checks that the proof is valid. The famous PCP Theorem provided a characterization of the NP complexity class that demonstrated that all statements in NP could be verified in polylogarithmic time, and recent improvements leveraging interactive oracle proofs have made these proofs practical.

In this case, because we want to support the migration of all existing Solana dApps to our rollup, we must implement the entire eBPF virtual machine as a zero-knowledge circuit. A zero-knowledge virtual machine (ZK VM) is capable of running any program. For a ZK VM, the entire virtual machine is implemented as a circuit, and the *program* is given as input, a Von Neumann machine. [citation]



There are two approaches to building ZK VMs: develop a bespoke VM for the purpose of zero-knowledge programs (such as Cairo or Miden [citation]) or take an existing VM and make it zero-knowledge, such as the Ethereum Virtual Machine or the eBPF. For the EVM, some zk-EVMs have taken the approach of compiling Solidity directly to a "friendlier" zk-EVM bytecode, such as zkSync. [citation] The Solana execution layer uses the LLVM compiler infrastructure, so we can use LLVM to produce an intermediate representation (IR) of the program, which can subsequently be converted into an Executable and Linkable Format (ELF) file compatible with our ZK circuitry. [citation]


Compile program as input to ZK circuit via LLVM
ELF File
Circuit Representation
Arithmetization via PLONK
Polynomial Representation
Commitment (choose based on size tradeoff)
FRI
DARK
Kate commitments
Settlement

Our design is inspired by the RISC Zero and Scroll zk-EVM architectures [citation]. For our ZK VM, for each logical cycle, we would allow a single memory transaction to occur. In the diagram above, we specify that users would compile their code into an ELF file to be loaded by our ZK VM. This requires constraining and committing to the ELF file. At a high level, we load the ELF file into RAM, and then enter the program execution loop, where one at a time, each instruction is loaded, computed, and applies a state change until the program terminates. A more detailed explanation of how the resulting execution trace is handled is explained in the "Appendix".

Open questions:

- Succinct recursive proof compositions: We might look into systems such as SuperSonic, Halo 2, and Fractal, in addition to connecting with the Penumbra team.
- Hashing: Decide on plookup, Poseidon, and recent advancements.
- Are there any unexpected difficulties in implementing the native programs? Presumably the Solana Program Library (SPL) all compiles to BPF.
- How do we handle account dependencies given that they require dynamic dispatch on Solana, and therefore the state requirement is not known at compile-time?

Pros of the ZK rollup: STARKs have asymptotically nearly-optimal runtime. All users benefit from fast finality once the ZK proof has been confirmed by the settlement layer.

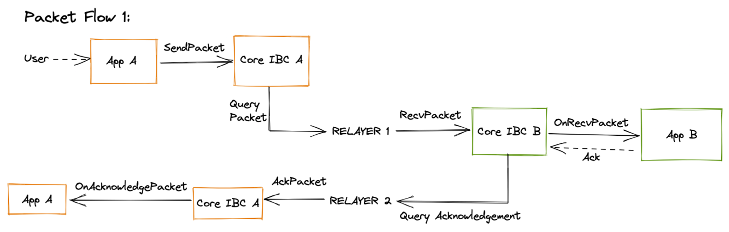
Cons of the ZK rollup: In the short run, the proving system is quite slow, taking hours to run. There are unanswered research questions with this approach.

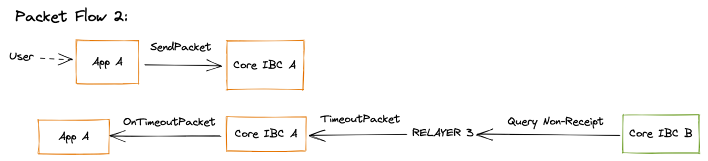
Inter-Blockchain Communication

Inter-blockchain communication (IBC) is the largest interoperability protocol, connecting blockchains within the Cosmos ecosystem. For the month of June 2022, there were over 2.5 million IBC transfers with a total volume over $500 million. IBC only specifies the messaging layer (data transport, authentication, and ordering), while the application layer is left for developers to define.

The messaging layer requires implementing three steps:

1. A handshake between chains, which facilitates authentication
2. Verifying data packets between chains, which requires the use of light clients
3. Defining routing options including timeouts and ordering guarantees





To facilitate (2) for the Solana execution layer, we must build light clients that provide consensus state introspection: visibility into the current block height, consensus state, and a bounded number of past headers. The Solana state hash doesn't provide proofs of inclusion (or non-inclusion), so we might modify the Solana state storage.

We implement inter-blockchain communication by forking the Penumbra Rust code, which is built on top of the IBC crate maintained by Informal Systems. The IBC messages must be subject to a challenge period for the optimistic rollup implementation.

Tokenomics

Since many L2s have yet to launch their tokens, L2 tokenomics is still relatively unexplored. The tokenomics must align the incentives of various parties: sequencers, provers, token stakers, protocol users, and liquidity providers.

We propose the following non-exhaustive list of avenues for the Eclipse token to accrue value:

- Burning a portion of sequencer or prover fees and/or returning to stakers
- Capturing and returning maximal extractable value (MEV) to stakers
- Incubating dApps built on Eclipse to share value at the application layer
- Supporting enterprise app chains
- Participating in on-chain governance
- Act as the currency to pay for interchain security for Eclipse sequencers
- Generating yield from the locked assets in Eclipse rollups

- Charging fees for app chain deployment
- Bridging fees
- Offering a default wallet with fees

We might also incorporate an inflationary schedule and ecosystem incentives such as activity rewards, contributor rewards, and airdrops.

Appendix

PLONKish Arithmetization: Execution Trace

For the ZK VM solution, an execution trace is generated for each program. Each instruction execution would get multiple rows in the execution trace, such as below. Motivated by the Scroll zk-EVM design, we identify the start of an instruction with a selector polynomial S and include data in columns C_i.

S (selector)	**C_1 (column 1)**	**C_2**	**C_3**
…			
1	pc (program counter)	g (gas)	Other context…
0	1 (add opcode)	0 (mul opcode)	Other opcodes…
0	a (first input)	b (second input)	c (output)
1	…	…	…

The above segment of the execution trace is a simplified representation of an addition instruction. One type of constraint enforces that each opcode cell is either exactly 0 or 1. Below is how we encode this constraint for the addition opcode:

$$S(x) * [C_1(\omega * x) * (1 - C_1(\omega * x))] = 0$$

We also enforce that exactly one opcode is turned on:

$$S(x) * [1 - (C_1(\omega * x) + \ldots + C_n(\omega * x))] = 0$$

Enforcing that the addition was executed correctly looks like this:

$$S(x) * C_1(\omega * x) * (a + b - c) = 0 \rightarrow$$
$$S(x) * C_1(\omega * x) * (C_1(\omega^2 * x) + C_2(\omega^2 * x) - C_3(\omega^2 * x)) = 0$$

We include additional constraints on state such as the program counter and gas.

Memory Transactions

Inspired by RISC Zero, during execution we record reads and writes to each memory address, generating a witness:

Cycle #	Address	Data	Read / Write
1	6	10	1 (write)
3	7	22	1
8	6	10	0
9	6	8	1

For verification, we sort this memory transaction trace by Address first, and Cycle second:

Cycle #	Address	Data	Read / Write
1	6	10	1
8	6	10	0
9	6	8	1
3	7	22	1

We enforce the same constraint as RISC Zero [citation] with the following pseudo-code:

```
if Address[n - 1] == Address[n]:
    assert Cycle[n] > Cycle[n - 1]
    if not Write[n]:
        assert Data[n - 1] == Data[n]
else:
    assert Address[n] > Address[n - 1]
```

We use PLONK to verify that the memory transaction trace is a permutation of the memory verification table.

Lookup Tables

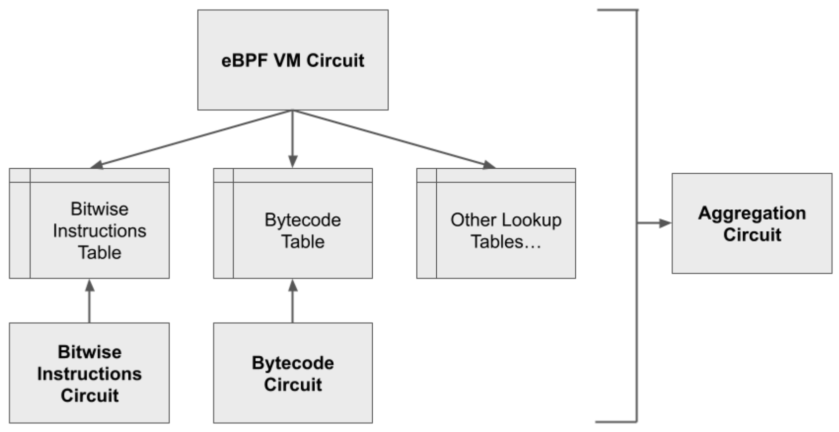
Some non-trivial constraints require plookup.

One such constraint is on the program counter and opcode cells. For a given program counter address, we must check the bytecode of the program to ensure the correct opcode cell is set to 1. We handle this by loading the bytecode into a table and performing a plookup.

Another non-trivial constraint might involve bitwise instructions such as XOR. We can check the XOR instruction against the following table:

C_1 (input 1)	C_2 (input 2)	C_3 (output)
0	0	0
0	1	1
1	0	1
1	1	0

These tables again must be constrained by circuits, where the proofs are composed by an aggregation circuit to be submitted for settlement:


eBPF VM Circuit
Bitwise Instructions Table
Bytecode Table
Other Lookup Tables…
Bitwise Instructions Circuit
Bytecode Circuit
Aggregation Circuit